Database Library Documentation

Table of Contents

Introduction

This library provides a Java-based abstraction layer for interacting with MySQL databases. It supports **CRUD operations**, **transactions**, **joins**, **subqueries**, **pagination**, and more. The library is designed to be easy to use while offering advanced features for complex queries.

Classes and Methods

Database Class

- **Purpose**: Manages database connections and transactions.
- **Methods**:
  - Database(String dbName, String user, String password): Constructor.
  - connect(boolean autoCommit): Connects to the database.
  - execute(DatabaseQuery query): Executes a query.
  - request(DatabaseQuery query): Executes a query and returns a ResultSet.
  - batchExecute(List<DatabaseQuery> queries): Executes multiple queries in a batch.
  - commit(): Commits the current transaction.
  - rollback(): Rolls back the current transaction.
  - setSavepoint(String name): Creates a savepoint.
  - rollback(Savepoint savepoint): Rolls back to a specific savepoint.
  - close(): Closes the database connection.

Condition Class
- **Purpose**: Represents SQL conditions (e.g., =, >, IN, BETWEEN).
- **Methods**:
    - Condition(Column column, ConditionType condition, Object value): Constructor for single-value conditions.
    - Condition(Column column, ConditionType condition, List<Object> values): Constructor for multi-value conditions (e.g., IN).
    - Condition(Column column, ConditionType condition, DatabaseQuery subquery): Constructor for subquery conditions.
    - toSQL(): Generates the SQL representation of the condition.
    - getParameterValues(): Returns the values for parameterized queries.

Join Class
- **Purpose**: Represents SQL joins (e.g., INNER JOIN, LEFT JOIN).
- **Methods**:
    - Join(Column from, Column on, JoinType type): Constructor.
    - toSQL(): Generates the SQL representation of the join.

DatabaseQuery Class
- **Purpose**: Base class for all query types.
- **Methods**:
    - build(): Abstract method to generate the SQL query.
    - setParameters(PreparedStatement stmt): Sets parameters for the query.

Insert Class
- **Purpose**: Represents an INSERT query.
- **Methods**:
    - Insert(Table table): Constructor.
    - column(Column column, Object value): Adds a column-value pair.
    - build(): Generates the SQL INSERT query.

Update Class
- **Purpose**: Represents an UPDATE query.
- **Methods**:
    - Update(Table table): Constructor.
    - set(Column column, Object value): Adds a column-value pair.
    - where(Condition... conditions): Adds conditions to the query.
    - build(): Generates the SQL UPDATE query.

Select Class

- **Purpose**: Represents a SELECT query.
- **Methods**:
  - Select(Table table): Constructor.
  - columns(Column... cols): Specifies columns to select.
  - where(Condition... conditions): Adds conditions to the query.
  - join(Join join): Adds a join to the query.
  - limit(int limit): Sets a limit on the number of rows returned.
  - offset(int offset): Sets an offset for pagination.
  - aggregate(String function, Column column, String alias): Adds an aggregate function (e.g., COUNT, SUM).
  - build(): Generates the SQL SELECT query.

Delete Class

- **Purpose**: Represents a DELETE query.
- **Methods**:
  - Delete(Table table): Constructor.
  - where(Condition... conditions): Adds conditions to the query.
  - build(): Generates the SQL DELETE query.

Table Class

- **Purpose**: Represents a database table.
- **Methods**:
  - Table(String name): Constructor.
  - Table(String database, String name): Constructor for tables in a specific database.
  - getName(): Returns the full table name (e.g., database.table).

Column Class

- **Purpose**: Represents a table column.
- **Methods**:
  - Column(String parent, String name, String alias): Constructor.
  - getFullName(): Returns the full column name (e.g., table.column).
  - getTableName(): Returns the table name.

SIMPLE USAGE

```
// Initialize database
Database db = new Database("mydb", "user", "password");

// Insert
Insert insert = new Insert(new Table("users"))
    .column(new Column("users", "id", null), 1)
    .column(new Column("users", "name", null), "John Doe");
db.execute(insert);
db.commit();

// Select
Select select = new Select(new Table("users"))
    .columns(new Column("users", "id", null), new Column("users", "name", null))
    .where(new Condition(new Column("users", "id", null), ConditionType.EQ, 1));
ResultSet rs = db.request(select);
while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
}

// Update
Update update = new Update(new Table("users"))
    .set(new Column("users", "name", null), "Jane Doe")
    .where(new Condition(new Column("users", "id", null), ConditionType.EQ, 1));
db.execute(update);
db.commit();
```

```
// Delete
Delete delete = new Delete(new Table("users"))
    .where(new Condition(new Column("users", "id", null), ConditionType.EQ, 1));
db.execute(delete);
db.commit();
```

ADVANCED USAGE

```
// Pagination
Select select = new Select(new Table("users"))
    .columns(new Column("users", "id", null), new Column("users", "name", null))
    .limit(10)
    .offset(20);

// Joins
Select selectWithJoin = new Select(new Table("users"))
    .join(new Join(new Column("users", "id", null), new Column("orders", "user_id", null), Join
Type.INNER));

// Subqueries
Select subquery = new Select(new Table("orders"))
    .columns(new Column("orders", "user_id", null))
    .where(new Condition(new Column("orders", "id", null), ConditionType.GT, 100));
Select selectSubquery = new Select(new Table("users"))
    .where(new Condition(new Column("users", "id", null), ConditionType.IN, subquery));

// Transactions with Savepoints
db.connect(false);
Savepoint savepoint = db.setSavepoint("SAVEPOINT_1");
db.rollback(savepoint);
db.commit();
```

Detailed Usage

1. Transactions with Savepoints

Savepoints allow you to roll back to a specific point within a transaction, rather than rolling back the entire transaction.

Example

```
try {
    // Insert a new user
    Insert insertUser = new Insert(new Table("users"))
        .column(new Column("users", "id", null), 101)
        .column(new Column("users", "name", null), "Alice");
    db.execute(insertUser);

    // Create a savepoint
    Savepoint savepoint = db.setSavepoint("SAVEPOINT_1");

    // Insert an order for the user
    Insert insertOrder = new Insert(new Table("orders"))
        .column(new Column("orders", "id", null), 1001)
        .column(new Column("orders", "user_id", null), 101)
        .column(new Column("orders", "amount", null), 50.0);
    db.execute(insertOrder);

    // Rollback to savepoint (undo the order insertion)
    db.rollback(savepoint);

    // Commit the transaction (only the user insertion will be saved)
    db.commit();
} catch (SQLException e) {
    db.rollback(); // Rollback the entire transaction on error
    e.printStackTrace();
} finally {
    db.close();
}
```

2. Batch Operations

Batch operations allow you to execute multiple queries in a single round-trip to the database, improving performance.

Example

```
Database db = new Database("mydb", "user", "password");

List<DatabaseQuery> batch = new ArrayList<>();
batch.add(new Insert(new Table("users"))
    .column(new Column("users", "id", null), 102)
    .column(new Column("users", "name", null), "Bob"));
batch.add(new Insert(new Table("users"))
    .column(new Column("users", "id", null), 103)
    .column(new Column("users", "name", null), "Charlie"));

db.batchExecute(batch);
db.commit();
db.close();
```

3. Subqueries

Subqueries allow you to nest one query inside another, enabling complex filtering and data retrieval.

Example

```
Database db = new Database("mydb", "user", "password");

// Subquery: Select user IDs from the orders table where the amount is greater than 100
Select subquery = new Select(new Table("orders"))
    .columns(new Column("orders", "user_id", null))
    .where(new Condition(new Column("orders", "amount", null), ConditionType.GT, 100));

// Main query: Select users whose IDs are in the result of the subquery
Select select = new Select(new Table("users"))
    .columns(new Column("users", "id", null), new Column("users", "name", null))
    .where(new Condition(new Column("users", "id", null), ConditionType.IN, subquery));

ResultSet rs = db.request(select);
while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
}

db.close();
```

4. Joins

Joins allow you to combine data from multiple tables based on related columns.

Example

```
Table users = new Table("users");
Table orders = new Table("orders");

Select select = new Select(users)
    .columns(
        new Column("users", "id", "user_id"),
        new Column("users", "name", "user_name"),
        new Column("orders", "id", "order_id"),
        new Column("orders", "amount", "order_amount")
    )
    .join(new Join(
        new Column("users", "id", null),
        new Column("orders", "user_id", null),
        JoinType.INNER
    ))
    .where(new Condition(new Column("users", "id", null), ConditionType.EQ, 101));

ResultSet rs = db.request(select);
while (rs.next()) {
    System.out.println(
        "User ID: " + rs.getInt("user_id") + ", " +
        "Name: " + rs.getString("user_name") + ", " +
        "Order ID: " + rs.getInt("order_id") + ", " +
        "Amount: " + rs.getDouble("order_amount")
    );
}

db.close();
```

5. Pagination

Pagination allows you to retrieve data in chunks, which is useful for large datasets.

Example

```
Database db = new Database("mydb", "user", "password");

Select select = new Select(new Table("users"))
    .columns(new Column("users", "id", null), new Column("users", "name", null))
    .limit(10) // Fetch 10 records
    .offset(20); // Skip the first 20 records

ResultSet rs = db.request(select);
while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
}

db.close();
```

6. Aggregate Functions

Aggregate functions like COUNT, SUM, AVG, etc., allow you to perform calculations on sets of rows.

Example

```
Database db = new Database("mydb", "user", "password");

Select select = new Select(new Table("orders"))
    .aggregate("SUM", new Column("orders", "amount", null), "total_amount")
    .where(new Condition(new Column("orders", "user_id", null), ConditionType.EQ, 101));

ResultSet rs = db.request(select);
if (rs.next()) {
    System.out.println("Total Amount: " + rs.getDouble("total_amount"));
}

db.close();
```

7. Complex Conditions

You can combine multiple conditions using AND or OR for advanced filtering.

Example

```
Database db = new Database("mydb", "user", "password");

Select select = new Select(new Table("users"))
    .columns(new Column("users", "id", null), new Column("users", "name", null))
    .where(
        new Condition(new Column("users", "id", null), ConditionType.GT, 100),
        new Condition(new Column("users", "name", null), ConditionType.LIKE, "A%")
    );

ResultSet rs = db.request(select);
while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
}

db.close();
```

8. Dynamic Query Building

You can dynamically build queries based on runtime conditions.

Example

```
Database db = new Database("mydb", "user", "password");

Select select = new Select(new Table("users"))
    .columns(new Column("users", "id", null), new Column("users", "name", null));

// Add conditions dynamically
if (someCondition) {
    select.where(new Condition(new Column("users", "id", null), ConditionType.GT, 100));
}

ResultSet rs = db.request(select);
while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
}

db.close();
```

9. Handling Large Result Sets

For large result sets, use a LIMIT and OFFSET loop to process data in chunks.

Example

```
Database db = new Database("mydb", "user", "password");

int limit = 100;
int offset = 0;

while (true) {
    Select select = new Select(new Table("users"))
        .columns(new Column("users", "id", null), new Column("users", "name", null))
        .limit(limit)
        .offset(offset);

    ResultSet rs = db.request(select);
    if (!rs.next()) break; // Exit loop if no more rows

    do {
        System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
    } while (rs.next());

    offset += limit;
}

db.close();
```

10. Combining Features

You can combine multiple advanced features in a single query.

Example

```
Database db = new Database("mydb", "user", "password");

// Subquery: Select user IDs from the orders table where the amount is greater than 100
Select subquery = new Select(new Table("orders"))
    .columns(new Column("orders", "user_id", null))
    .where(new Condition(new Column("orders", "amount", null), ConditionType.GT, 100));

// Main query: Select users whose IDs are in the result of the subquery, with pagination
Select select = new Select(new Table("users"))
    .columns(new Column("users", "id", null), new Column("users", "name", null))
    .where(new Condition(new Column("users", "id", null), ConditionType.IN, subquery))
    .limit(10)
    .offset(20);

ResultSet rs = db.request(select);
while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
}

db.close();
```